Wrappers for ADOL-C in scripting languages using SWIG

Kshitij Kulshreshtha*and Sri Hari Krishna Narayanan†

March 2016

1 Introduction

R is a language and environment for statistical computing and graphics [1]. It currently is widely used in statistics and data mining. To obtain derivatives in R, one can use several non-native approaches, including the TMB system [2] and Ryacas [3]. However, none of these options support the differentiation of functions expressed as R programs, as would an algorithmic differentiation (AD) tool for R. Attempts to develop such a tool include radx [4]. This tool is capable of computing first- and second-order forward-mode derivatives of univariate functions. But it is no longer actively developed. Natively, inside R, the `numderiv` package provides methods for calculating (usually) accurate numerical first and second order derivatives [5]. Accurate calculations are done by using Richardson's extrapolation, or, when applicable, a complex step derivative is available. A simple difference method is also provided. The `deriv` function from the `stats` package computes derivatives of simple expressions, symbolically [6]. Because numerical differences cannot be reliably accurate and cannot compute adjoints, there is a need to provide derivatives within R using AD tools.

One method to obtaining derivatives is ADOL-C [7]. It is a mature and widely applied tool for algorithmic differentiation using operator overloading in the C++ language. Because of the language dependency it can natively be used only with applications that were originally written in C or C++. Previously, a Python-wrapper [8, 9] was written for the most widely used functionality in ADOL-C. It can be used to compute first and higher order derivatives in both forward and reverse mode for applications written in Python. This wrapper was written manually, however, and must be maintained and updated manually to keep in sync with the changes and new features of the C++ library. Also only the most commonly used ADOL-C API calls were available.

Given the success of manually interfacing Python with ADOL-C we have investigated an automated interfacing mechanism for ADOL-C with R and Python. We used the SWIG interface generator for this purpose. Using the interfaces that were generated, we are able to use ADOL-C from within R and Python to obtain derivatives. The rest of the document shows how SWIG was used and provides examples of ADOL-C usage.

2 SWIG interface generator

SWIG is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages. SWIG is typically used to parse C/C++ interfaces and generate the "glue code" required for the target languages to call into the C/C++ code [10, 11, 12, 13]. It can generate interfaces for many different languages including R, Python, TCL, and Octave. Important for this work, by using SWIG, an interface for ADOL-C can be generated automatically during the build process of the ADOL-C library. Once the interface generation with SWIG has been set up correctly for the intended target languages, the generated interface will automatically contain all the new features and updates from ADOL-C.

SWIG generates interfaces based on an input file (usually `somemodule.i`). This input file consists of SWIG macros. A simple module may be defined, by using the input file in Figure 1. This will create a module with the name `mymodule`

(a)

```
%module mymodule
%{
#include <myheader.h>
%}

%include <myheader.h>
```

(b)

```
%module adolc
%{
#include <adolc/adolc.h>
%}
// many ignore and renames
%include "adolc_all.hpp" // generated by running
     // C++ preprocessor
```

Figure 1: (a) SWIG input file for an simple example module; (b) skeleton ADOL-C SWIG input file

containing a wrapped interface in the scripting language of choice for the C/C++ API declared in the file that is given

*Paderborn University, Germany, kshitij@math.upb.de
†Argonne National Laboratory, USA, snarayan@mcs.anl.gov

in the `%include` macro. In this case it is `<myheader.h>`. Actual C/C++ code is given between the macro delimiters `%{` and `%}`. This code is required to compile and link the generated interface with the original C/C++ library. Other macros of importance are `%ignore` and `%rename`. These will cause SWIG to ignore a certain C/C++ API name or rename it to something else for the generated interface, respectively. This feature is useful if these names contain certain characters that are unsupported by the target language or include keywords or if wrapping these in the target language is not desirable at all.

One caveat of the `%include` macro is that it will read only the named file and will not recurse into any files that are `#included` inside it, unlike the C/C++ preprocessor. This is a challange for processing ADOL-C via SWIG, since the outer header file `<adolc/adolc.h>` contains a large number of `#include` directives for subsidiary headers, as well as system headers. Running the C++ preprocessor directly results in a file containing all the APIs from all the system headers as well as all the subsidiary headers. We do not need to wrap the system APIs for the target language, only the ADOL-C API. We therefore wrote a Python script that first excludes all the system headers from the ADOL-C headers and then runs the C++ preprocessor on it to produce a flat single header containing all ADOL-C APIs, but no system APIs. This file is then `%included` and processed with SWIG, and then the generated sources are compiled.

R interface The expectation that SWIG would generate a working interface from the input file automatically was not met. We encountered several difficulties when R was the chosen target language. First, the generated interface for R contained inplace modification of arrays given as arguments. Generally, R programmers prefer to use the returned values from a function as the output instead of modifying the input arguments. However, this is the standard practice in C/C++ when multiple values need to be output. SWIG version 3.0.8 did not have the necessary mechanism for modifying the input arguments. We therefore needed to modify the SWIG sources themselves and introduced `%typemap(argout)` instructions as detailed in Section 11.5 of the SWIG Documentation for considering 1D and 2D arrays as inplace modifiable arguments in R. These changes in the SWIG sources are not yet, at the time of writing this, included in any official SWIG release or source repository.

Another difficulty is imposed by the structure of the R language itself. It does not allow for operator overloading in the same sense as C++. The C++ compiler is responsible for choosing the correct operator based on context in any expression. In R, the programmer is responsible for checking the arguments to any overloaded function or operator and dispatching the correct version. As a safeguard against inadvertent overloading of common mathematical operators, the SWIG-generated interface contains named functions for such operators (e.g. "`Plus`" for `operator +`). To utilize operator overloading correctly, we needed to modify the generated R source code as shown in Figure 2

(a)

```
'Plus' <- function(...) {
  argtypes <- mapply(class, list(...));
  argv <- list(...);
  argc <- length(argtypes);
# dispatch functions
  if (argc == 1) {
    if (extends(argtypes[1], '_p_badouble') && length(
      argv[[1]]) == 1) {
      f <- Plus__SWIG_0;
    }
  } else if (argc == 2) {
    if (extends(argtypes[1], '_p_badouble') && length(
      argv[[1]]) == 1 && extends(argtypes[2], '_p_
      badouble') && length(argv[[2]]) == 1) {
      f <- Plus__SWIG_1;
    } else {
      # many such cases ...
    }
  } else {
    stop("cannot find overloaded function for Plus with
      argtypes (",toString(argtypes),")");
  };
  f(...);
}
```

$\Longrightarrow$

(b)

```
'oldplus' <- '+'
'+' <- function(...) {
  argtypes <- mapply(class, list(...));
  argv <- list(...);
  argc <- length(argtypes);
# dispatch functions
  if (argc == 1) {
    if (extends(argtypes[1], '_p_badouble') && length(
      argv[[1]]) == 1) {
      f <- Plus__SWIG_0;
    } else {
      f <- oldplus;
    }
  } else if (argc == 2) {
    if (extends(argtypes[1], '_p_badouble') && length(
      argv[[1]]) == 1 && extends(argtypes[2], '_p_
      badouble') && length(argv[[2]]) == 1) {
      f <- Plus__SWIG_1;
    } else if ( # other argument checks ) {
      # many such cases ...
    } else {
      f <- oldplus;
    }
  } else {
    f <- oldplus;
  };
  f(...);
}
```

Figure 2: (a) Generated R interface source, (b) manual modification for operator overloading

Python and NumPy interface Using the experience in creating an interface between R and ADOL-C, we were able to create an interface for ADOL-C and Python. SWIG has been used extensively to generate Python interfaces to C++ software, for example, in the FEniCS project [14, 15, 16]. There are differences, however, in the way Python deals with intermediate results to those in C++, as well as how array data structures are handled in NumPy, the numerical mathematics module in Python. In C++, the assignment operator can be overloaded to account for the temporary intermediate `adub` objects that are allocated on the stack with short lifetimes. In Python, the assignment operator cannot be overloaded, and all objects must be allocated on the heap. This difficulty is straightforward to

handle; we can simply `%ignore` the operators defined in C++ and write simple one-line wrappers that will return a heap allocated `adub*` instead of a stack-allocated `adub` using a special typecast operator defined in ADOL-C. Python's own garbage collection mechanism deals with the resulting memory.

Arrays in Python are handled as `numpy.array` or `numpy.ndarray` objects. The NumPy authors have provided a SWIG input file `numpy.i` containing the specific typemaps for converting a C/C++ array argument given as a pointer and its size in a separate function argument. However, these work only if each such array has its own size right next to it. In ADOL-C, most drivers take several array arguments with the size; either the number of dependents or number of independents, and these sizes are known from the trace. For all such functions to be able to interpret and return NumPy arrays properly, some simple wrappers are again required, with modified C++ signatures. A few such signatures are shown in Figure 3. These wrappers are written purely in C/C++, and the maintainer does not need to write any Python code or use any Python or NumPy API for C.

(a)

```
int gradient(short tag, int n,
             double* x,    // size n
             double* g);   // size n, g preallocated
int jacobian(short tag, int m, int n,
             double* x,    // size n
             double** J);  // size m, n, J preallocated
int vec_jac(short tag, int m, int n,
            double* x,     // size n
            double* v,     // size m
            double* w);    // size n, w preallocated
```

(b)

```
void npy_gradient(short tag,
                  double* x, int n0,
                  double** g, int* n1); // *n1 = n, from trace
                                        // g allocated & returned to python
void npy_jacobian(short tag,
                  double* x, int n0,
                  double** J, int* m1, int* n1);
                                        // *m1 == m && *n1 = n, from trace
                                        // J allocated & returned to python
void npy_vec_jac(short tag, int repeat,
                 double* x, int n0,
                 double* v, int m1,
                 double** w, int* n1);  // *n1 = n, from trace
                                        // w allocated & returned to python
```

(c)

```
g = gradient(tag,x)
J = jacobian(tag,x)
w = jac_vec(tag,repeat,x,v)
```

Figure 3: (a) ADOL-C drivers with original signatures; (b) their NumPy array-aware wrapper signatures; (c) their usage in Python

3 Using the generated interfaces

Figure 4(a) shows the example usage of ADOL-C from within R. After the initial loading of the ADOL-C dynamic library, the code mimics familiar ADOL-C drivers written in C++. The main differences are the use of special SWIG-created interface functions for identifying the independent and dependent variables and the use of the `adbouble` function to initialize the independent variables. The tracing portion of the code is used to create a trace of the computation. Following that, the different functions that are invoked use the trace to compute forward or reverse derivatives. The same example in Python is shown is Figure 4(b).

4 Conclusion

We have used SWIG to automatically create an interface between ADOL-C and R as well as ADOL-C and Python. In the future, we will add support for computing the derivatives of sparse derivatives in R. Within R, we will study the usage of derivatives obtained through ADOL-C in the context of solving optimization problems and machine learning.

Acknowledgments. This work was funded in part by a grant from DAAD Project Based Personnel Exchange Programme and by support from the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357. We thank Prasanna Balaprakash, Joseph Wang, and Richard Beare for their suggestions.

References

[1] The R project for statistical computing. `http://www.r-project.org/`, 2014.

[2] ADMB - Template Model Builder. `http://www.admb-project.org/developers/tmb`, 2014.

[3] Rob Goedman, Gabor Grothendieck, Søren Højsgaard, and Ayal Pinkus. Ryacas - An R Interface to the YACAS Computer Algebra System. `http://cran.r-project.org/web/packages/Ryacas/vignettes/Ryacas.pdf`, 2014.

[4] Automatic Differentiation in R. `https://github.com/quantumelixir/radx`, 2014.

[5] Package numDeriv. `https://cran.r-project.org/web/packages/numDeriv/numDeriv.pdf`, 2015.

(a)

```
dyn.load(paste("adolc", .Platform$dynlib.ext, sep=""))
source("adolc.R")
cacheMetaData(1)

trace_on(1)
a <- adouble(2.0)
b <- adouble(1.0)
badouble_declareIndependent(a)
badouble_declareIndependent(b)
x <- a*a + b*b + 2 *a *b
badouble_declareDependent(x)
trace_off()

c1 <- c(1.0,2.0)
c2 <- c(0.0)
zos_forward(1,1,2,1,c1,c2)

c4 <- c(1.0, 1.0)
c5 <- c(0.0)
fos_forward(1,1,2,1,c1,c4,c2,c5)

c6 <- c(1.0, 2.0)
c7 <- c(0.0, 0.0)
gradient(1,2, c6, c7);

c8<- c(1.0)
c9 <- c(0.0,0.0)
fos_reverse(1,1,2,c8,c9)

c10 <- c(1.0, 2.0)
c11 <- matrix(0.0, ncol = 2, nrow = 1)
jacobian(1,1,2,c10,c11);

c12 <- c(1.0, 2.0)
c13 <- matrix(0.0, ncol = 2, nrow = 2)
hessian(1,2,c12,c13);
```

(b)

```
from adolc import *
import numpy as np

trace_on(1)

a = adouble()
b = adouble()
a <<= 2.0
b <<= 1.0
x = a*a + b*b + 2*a*b
x.declareDependent()
trace_off()

c1 = [1.0,2.0]
c2 = zos_forward(1,1,2,1,c1)
# c2 is np.array with .shape = (1,)

c4 = [1.0, 1.0]
(c2,c5) = fos_forward(1,1,2,1,c1,c4)
# c2 is np.array with .shape = (1,)
# c5 is np.array with .shape = (1,)

c7 = gradient(1,c1)
# c7 is np.array with .shape = (2,)

c8 = [1.0]
c9 = fos_reverse(1,1,2,c8)
# c9 is np.array with .shape = (2,)

c11 = jacobian(1,c1)
# c11 is np.array with .shape = (1,2)

c13 = hessian(1,c1)
# c13 is np.array with .shape = (2,2)
```

Figure 4: Example usage of ADOL-C from within (a) R and (b) python

[6] Symbolic and Algorithmic Derivatives of Simple Expressions. `https://stat.ethz.ch/R-manual/R-devel/library/stats/html/deriv.html`, 2016.

[7] Andrea Walther and Andreas Griewank. Getting started with ADOL-C. In Uwe Naumann and Olaf Schenk, editors, *Combinatorial Scientific Computing*. Chapman-Hall, 2012.

[8] Sebastian F. Walter. Algorithmic Differentiation in Python with PYADOLC and PYCPPAD. EuroScipy Conference, Leipzig, Germany, July 2009.

[9] Sebastian F. Walter. AD in Python with Application in Science and Engineering. Eighth EuroAD Workshop, The Numerical Algorithms Group, Oxford, UK, July 2009.

[10] Swig website. `http://www.swig.org/`, 2016.

[11] David M. Beazley. SWIG : An Easy to Use Tool for Integrating Scripting Languages with C and C++. 4th Annual Tcl/Tk Workshop, Monterey, CA, July 1996.

[12] David M. Beazley. Using SWIG to Control, Prototype, and Debug C Programs with Python. 4th International Python Conference, Livermore, CA, June 1996.

[13] David M. Beazley and Peter S. Lomdahl. Feeding a Large-scale Physics Application to Python. 6th International Python Conference, San Jose, CA, October 1997.

[14] M. S. Alnæs, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. E. Rognes, and G. N. Wells. The FEniCS Project Version 1.5. *Archive of Numerical Software*, 3(100), 2015.

[15] A. Logg and G. N. Wells. DOLFIN: Automated Finite Element Computing. *ACM Transactions on Mathematical Software*, 37(2), 2010.

[16] A. Logg, G. N. Wells, and J. Hake. DOLFIN: a C++/Python Finite Element Library. In *Automated Solution of Differential Equations by the Finite Element Method*, volume 84 of *Lecture Notes in Computational Science and Engineering*, chapter 10. Springer, 2012.